Report

Based on your solution and understanding of the assignment and lecture slides, fill out the questions below:

1. Measure the overall execution time of your MPI program, and measure the time required to collect data on rank 0 for verification (discuss differences for different input arguments). Explain how you measured execution time in the MPI code, and which values you used to print the elapsed time on rank 0. Why these values correctly represent the execution time of your program? Is the time required to collect data different when the program is executed on a single node?

Both the overall execution time of the MPI program and the time required to collect data on rank 0 are measured using **MPI_Wtime()** function. The execution time of the entire MPI program is measured by recording the start time (time_1) before the main computation begins and the end time (time_2) after the computation completes. The difference between these two times gives the total elapsed time.

The time required to collect data on Rank 0 for verification is measured separately but in similar manner, by recording the start time (time_3) before the MPI_Gatherv operation and the end time (time_4) after the operation completes. The difference between these two times gives the time required for data collection.

In a single-node execution, the data collection may be faster as the communication occurs within the memory of the same node. However, in a multi-node setup, the data needs to be transferred across network links, which can introduce additional delays and increase the data collection time.

2. How is the data distributed among the MPI processes in this example? How big is each matrix on each process with respect to the M and N? How did you handle the case where `(M%numprocs != 0)`? Does this have an effect on the computational kernel (the for-loops), and did you have to make any adjustments?

In this implementation, the code divides the matrix row-wise among the available MPI processes. Each process gets a portion of the rows to work on and the number of rows per process is initially calculated as M/size where size is the number of MPI processes. This is the base number of rows each process works on.

The remainder of M/size is distributed among the first M%size processes, giving them one extra row each to ensure all rows are processed.

For the purpose of boundary communication between adjacent processes, each local matrix includes additional "halo" rows. Each process, except those at the very top or bottom of the global matrix, includes one halo row at the top and one at the bottom, making the total local matrix size local_M_with_halo x N.

The uneven distribution necessitates careful indexing to ensure computations are performed correctly, particularly near the boundaries of each process's portion of the matrix. To ensure that boundary conditions between adjacent processes are accurate, communication routines are implemented to handle halo row exchanges.

3. What data needs to be communicated between MPI processes and at which points in your code? Which MPI routines have you used to accomplish this?

The boundary values of the matrix need to be exchanged between MPI processes. To accomplish this communication, **MPI_Sendrecv** routine is used which is designed to prevent deadlocks by simultaneously handling sending and receiving operations.

4. Are there any points in the code where you need to be careful not to introduce a deadlock? If so, where?

When implementing MPI, the potential deadlock scenario can arise during the data exchange between neighboring processes. Deadlocks occur when two or more processes wait indefinitely for an event that can only be caused by one of the waiting processes. In this implementation, exchanging halo rows between neighboring processes is the most crucial point for potential deadlocks.

Deadlocks can also occur with collective operations (e.g., MPI_Gather, MPI_Gatherv, MPI_Allreduce), if not all processes participate in the collective operation as expected.

5. Briefly explain how you collected data to rank 0 at the end of your code for verification on rank 0. Which routines have you used and how? What data was relevant, and on which ranks?

The data collection at the end of the computation phase is achieved using **MPI_Gatherv**, with preparatory steps ensuring each process knows how much data to send and rank 0 knows where to place incoming data in the complete grid. The **MPI_Gatherv** call on all processes instructs them to send their relevant grid section to rank 0. The root process (rank 0) prepares **recvcounts** and **displs** arrays to correctly receive and place the incoming data sections into **bigU**, the matrix that will contain the complete grid.

6. Include a table with execution times and speedup of the MPI program for different configurations given in the slides. Speedup needs to be measured with respect to the sequential version (around 50, 112, and 5 seconds for different versions on ALMA nodes – see slides). If you have these in an Excel sheet, you can just attach it with your submission. Plot MPI single node configurations on a separate speedup graph and combine it with your OpenMP results (if you have them) for comparison and discuss the performance differences.

Execution Time (seconds)

#	Sequential	OpenMP	MPI
a	50	10	9
b	112	20	14
c	5	0.6	0.7

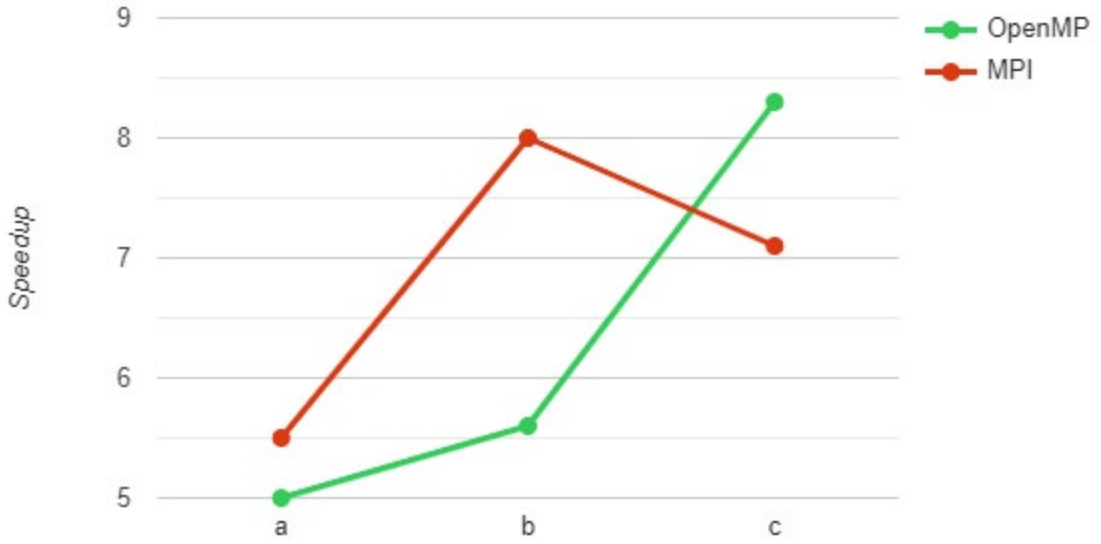
The table shows execution times for the following parameters:

a) OMP_NUM_THREADS=16 srun --nodes=1 ./a2-omp --m 2688 --n 4096 --epsilon 0.01 --max-iterations 1000
mpirun -np 16 ./a3-mpi --m 2688 --n 4096 --epsilon 0.01 --max-iterations 1000

b) OMP_NUM_THREADS=16 srun --nodes=1 ./a2-omp --m 2688 --n 4096 --epsilon 0.01 --max-iterations 2000
mpirun -np 16 ./a3-mpi --m 2688 --n 4096 --epsilon 0.01 --max-iterations 1500

c) OMP_NUM_THREADS=16 srun --nodes=1 ./a2-omp --m 1152 --n 1152 --epsilon 0.01 --max-iterations 1000
mpirun -np 16 ./a3-mpi --m 1152 --n 1152 --epsilon 0.01 --max-iterations 1000

I have not achieved the desired speedup for 4 nodes with MPI, and the verification is NOT OK.



7. How different are the MPI and the OpenMP code that you have developed? Which code is suitable for which type of architecture and how about the code complexity?

While both are aimed at improving performance by distributing the workload, they are designed for different hardware architectures and have different complexities in terms of coding and execution.

MPI is designed for distributed computing, where the program runs across multiple nodes in a cluster. Each process has its own local memory, and processes communicate by explicitly sending and receiving messages. This model is suitable for both shared and distributed memory systems but is primarily used for the latter.
OpenMP is used for shared memory architectures. It leverages threads within a single process that can access shared memory space. Parallelism is achieved by forking additional threads to handle parts of the computation and then joining them back.

MPI code tends to be more complex due to the explicit communication between processes. On the other hand, OpenMP code is often simpler and easier to write and understand because it requires fewer modifications to the serial code. Parallelism is achieved by adding directives that the compiler uses to parallelize the code automatically, reducing the burden of managing threads or processes directly.